VMM-based Hidden Process Detection and Identification using Lycosid

Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

Department of Computer Sciences, University of Wisconsin–Madison

{stjones,dusseau,remzi}@cs.wisc.edu

Abstract

Use of stealth rootkit techniques to hide long-lived malicious processes is a current and alarming security issue. In this paper, we describe, implement, and evaluate a novel VMM-based hidden process detection and identification service called Lycosid that is based on the cross-view validation principle. Like previous VMM-based security services, Lycosid benefits from its protected location. In contrast to previous VMM-based hidden process detectors, Lycosid obtains guest process information implicitly. Using implicit information reduces its susceptibility to guest evasion attacks and decouples it from specific guest operating system versions and patch levels. The implicit information Lycosid depends on, however, can be noisy and unreliable. Statistical inference techniques like hypothesis testing and linear regression allow Lycosid to trade time for accuracy. Despite low quality inputs, Lycosid provides a robust, highly accurate service usable even in security environments where the consequences for wrong decisions can be high.

Categories and Subject Descriptors D.4.6 [*Security and Protection*]

General Terms Design, Measurement, Security

Keywords Virtual Machine, Inference, Security

1. Introduction

Stealth rootkits that can hide processes are an important security issue. According to statistics gathered from Microsoft's *Malicious Software Removal Tool* [20], a significant fraction of the malware it encounters consists of stealth rootkits [22]. The ability to detect and respond to malicious hidden processes is a clear advantage in the race to defend network-attached computers.

In this paper we propose a VMM-based hidden process detection and identification service called Lycosid. Previously proposed VMM-based security services assume that the VMM has detailed *implementation* information about the guest operating systems they protect [10, 17]. In contrast, Lycosid is based on information implicitly obtained about guest operating systems. Because it does not depend on specific guest OS implementation details, Lycosid has two key advantages over previous approaches. First, it is more resilient to typical process hiding techniques, even those that manipulate and corrupt the privileged internal state of the OS kernel. Second, a single implementation within a VMM can be portable across very different operating systems like Windows and Linux, a fact we demonstrate in Section 7.

Like earlier approaches, Lycosid uses a technique called *cross-view* validation [30] to detect maliciously hidden OS processes. The technique works by observing a class of objects (*e.g.*, OS processes) from multiple perspectives and noting inconsistencies between views. One view, known as the untrusted view, is obtained from a high-level in the system. The other, known as the trusted view, is obtained from a low level that is unlikely to have been subverted by an attacker. If an object appears in the trusted view and does not appear in the untrusted view, the cross-view principle concludes that an object has been hidden. In this paper we follow the cross-view convention and use "trusted" to mean "more reliable" rather than "having formal security properties". The deeper within a system a trusted view can be obtained the better. Lycosid obtains its trusted view from deep within the system at the VMM-layer.

A VMM is an attractive place to deploy security monitoring services like anomaly detection systems [10, 13, 17]. By virtue of their location behind the relative security of the virtual machine interface, VMM-based services are better shielded from malicious attacks that originate from within a guest virtual machine [19], even if the guest operating system kernel is compromised. Though a VMM is separated from guests by a secure barrier, it still has ready access to the raw state of its guest virtual machines. For example, a VMM can easily read and write guest registers and memory and can observe guest I/O like disk and network requests.

While implicitly obtained information has the beneficial properties described above, it can be challenging to use effectively. For example, it can be noisy and is sometimes incorrect [15, 16]. Lycosid achieves accuracy by using statistical inference techniques like hypothesis testing and linear regression that trade time for accuracy. Despite low quality inputs, Lycosid provides a robust, highly accurate, and portable service usable even in security environments where the consequences for wrong decisions can be high.

Lycosid bases all of its detection and most of its identification decisions on *passively* obtained information. In some cases, however, we find that passive information is inadequate to reliably identify which of many candidate processes has been hidden. Lycosid introduces a new technique called *CPU inflation* that allows a VMM to influence the runtime of specific processes by carefully patching a process's executable code. Using CPU inflation, Lycosid can often transform a detectable, but unidentifiable, hidden process into a hidden process that can be reliably identified.

The rest of this paper is organized as follows. We first review process hiding techniques in Section 2. In Sections 3 and 4 we discuss the techniques used by Lycosid to detect and identify hidden processes. Section 5 discusses the evasion problem. We describe our implementation of Lycosid in Section 6 and Section 7 contains

VEE '08 March 5–7, 2008, Seattle, Washington, USA.

a detailed evaluation of the accuracy and performance of the implementation. We discuss related work in Section 8 and conclude in Section 9.

2. Process Hiding

When a system is compromised, it is common for an attacker to leave programs behind that advance the attacker's goals. This approach is especially favored when the attacker accesses a machine from a remote location over a network. For example, an attacker will often leave behind a back door program that listens to the network and allows the attacker to regain a privileged presence on a compromised system without re-exploiting a vulnerability [27]. In other cases, key capture or file system scanning programs are left running to collect additional useful information like login names, passwords, and financial records.

The presence of unexplained processes, network connections, or files is an indicator to a system administrator or intrusion detection system that a successful attack has occurred. To avoid tipping off a defender, an attacker will often attempt to hide their malicious processes, network connections or data files [2]. Hiding is typically accomplished by modifying some aspect of the system using a suite of tools called a stealth rootkit. For example, some rootkits modify program binaries like `ps`, `netstat`, and `ls` [21]. Other rootkits hook into the call path between a user application and the kernel by modifying libraries, dynamic linker structures, system call tables, or operating system functions that report system status [12]. Finally, some rootkits manipulate kernel data structures using so-called direct kernel object manipulation (DKOM) [9]. Rootkit hooks and modified kernel data structures lead to corrupted results of user requests, effectively hiding the presence of malicious resources [3, 28]. The list of techniques available to hide system resources is growing.

Long lived malicious processes are the most likely candidates for hiding. The probability of detecting a short lived malicious process via a process introspection tool like `ps` is relatively small, so an attacker rarely goes to the trouble of hiding a short-lived process. The long-lived nature of maliciously hidden processes has implications for the kinds of detection techniques that are feasible.

3. Detection

The Lycosid service is partitioned into detection and identification components. We discuss the detection component in this section. Detection consists of determining if any processes running within a guest virtual machine have been hidden. The detection algorithm does not identify which processes are hidden. Identification is discussed in Section 4.

3.1 Approach

If a process has been hidden using any of the methods described in Section 2, it will not appear on a user-level process listing. It will, however, appear on a suitably obtained, low-level process list. Hence, to detect a hidden process we can compare the lengths of process lists obtained at a low (trusted) and a high (untrusted) level. If the trusted list is longer than the untrusted list we can conclude that at least one process has been hidden.

On an idle system, simply obtaining a single instance of the two process lists and comparing them would suffice to detect hidden processes. On an active system, however, where processes are being created and destroyed, the situation becomes more complicated. For example, Lycosid cannot perfectly synchronize the times at which it makes its two process list observations, so they may reflect different process-related states of the system. Additionally, the measurements taken within the VMM can be delayed, further complicating the inference. As the system experiences higher levels of process creation and exit activity, the problem worsens.

The Lycosid detection phase overcomes these issues by obtaining many pairs of measurements over time and performing a series of paired-sample hypothesis tests [25]. Each pair consists of a process count obtained from within the VMM and a process count obtained from within the guest. Using a hypothesis test, we can determine if the two process lists differ in length even when the system process state is in dramatic flux. The test procedure also provides the ability to quantitatively limit the chance that we assert one or more processes are hidden when in fact no hiding is taking place, *i.e.*, the false positive rate can be explicitly controlled.

Formally, let T be the length of the trusted process list and let U be the length of the untrusted process list. Our null and alternative hypotheses are then:

$$H_0 : T - U \leq 0 \tag{1}$$

$$H_1 : T - U > 0 \tag{2}$$

We use the non-parametric Wilcoxon rank-sign statistic [25] in our tests because it makes no assumptions about the distribution of the population from which our samples are drawn. Data analysis indicates that the distribution of $T - U$ is quite symmetric, but can be slightly skewed and is not normally distributed.

If we can reject the null hypothesis H_0 in favor of the alternative hypothesis H_1 at an appropriate level of confidence, we can quantitatively conclude that one or more processes is being hidden. The hypothesis test p-value indicates the probability of a false positive, *i.e.*, indicating hiding when the null hypothesis H_0 (no processes are hidden) is true. As with most anomaly detection systems, the consequences for false positives in the detection performed by Lycosid are significant. Too many false positives degrade confidence in the system and render the information it provides less valuable. Hence, we choose a conservative threshold confidence value ($\alpha = 2 \times 10^{-6}$). If the one-sided p-value computed during the hypothesis test falls below α, Lycosid reports that one or more processes have been hidden.

In addition to a hidden process indicator, the average difference observed between the two lists during the detection phase provides an estimate of the number of processes that have been hidden. This point estimate is used as input to the hidden process *identification* algorithm described in Section 4.

3.2 Details

Lycosid obtains a trusted view of guest processes from within a VMM. The VMM-based approach has advantages over any technique that obtains trusted information from within the guest itself because a VMM is typically much harder to subvert than guest software services or even the guest operating system kernel. This fact follows from the relatively smaller and well-defined virtual machine interface that separates the guest from the VMM.

VMI [10], for example, uses this advantage to provide various resilient security services within a VMM, one of which is hidden process detection. Lycosid differs from VMI in the way it obtains trusted information about the guest operating system. VMI exploits detailed information about the location and semantics of private Linux kernel data structures to obtain a low-level guest process list. In contrast, Lycosid obtains its low-level guest information implicitly. This is a key advantage of Lycosid. No detailed implementation information about the guest is required. As a result, Lycosid can be deployed without taking versions and patch levels of the target operating systems into account. Our single implementation of Lycosid within a VMM, for example, can support multiple versions of both Windows and Linux without modification.

Lycosid uses Antfarm [15] to obtain its trusted view of guest operating system processes. Antfarm is a VMM component that implicitly obtains information about guest operating system events like process creations and exits by observing closely related events like virtual address space creation and destruction. Antfarm can also provide estimates of other process-related quantities like CPU time consumed, working set size, and context switch counts by observing their virtual address space analogues.

Lycosid obtains its untrusted view of guest operating system processes the same way that VMI does. A network connection is made from the VMM to the guest and a user-level program within the guest is invoked to enumerate processes. On a UNIX-like system the `ps` command can provide this information. On Windows systems, various utilities like `pslist.exe` [4] or the built-in `tasklist.exe` can be used.

Lycosid obtains trusted and untrusted process lists at short random intervals. A window of the most recent samples is preserved for use in hypothesis testing. The size of the window and the sample interval are configurable. In our implementation, samples are obtained every one second on average. Up to the most recent 600 samples are used in each hypothesis test. Approximately every minute, we test the null hypothesis that the two lists are the same length. Given the detection threshold $\alpha = 2 \times 10^{-6}$, our configuration corresponds to about one expected false positive per year.

4. Identification

After detecting that one or more processes have been hidden, the natural next step is to *identify* which processes have been hidden. Identifying specific hidden processes enables a more effective VMM response to the malicious activity.

Given only the information provided by the hiding detector, each process visible from within the VMM is equally likely to be the culprit. Our approach for identifying which processes have been hidden is to select a measurable quantity associated with hidden processes and use it to choose from the set of candidate processes.

4.1 Approach

As a process executes, it consumes CPU time. Both the operating system and a process-aware VMM like Lycosid can account CPU time to specific processes. Let G_i denote the CPU time for process i as observed from within a guest. Let V_j be the CPU time accumulated by process j as seen by the VMM. Then, when hiding occurs, the quantity

$$H = \sum_j V_j - \sum_i G_i \tag{3}$$

represents the total CPU time observed within the VMM that is not accounted for by processes visible to the guest, *i.e.*, it is the CPU time used by hidden processes. We can construct a linear equation using H and the per-process CPU times we have obtained from within the VMM.

$$H = \beta_1 V_1 + \beta_2 V_2 + ... + \beta_n V_n \tag{4}$$

Equation 4 holds if the coefficients β_j take the value 1 for processes that are hidden and 0 for non-hidden processes. We can identify likely hidden processes by fitting a multiple variable linear model using least-squares regression on Equation 4 and choosing the N variables from the model that best explain the variance observed in H, where N is the estimated number of hidden processes obtained during the detection phase. Hence, we treat hidden process identification as a multiple linear regression variable selection problem.

```
# VMM PID      VMM proc runtime (s)
0x3a40         1.219
0xad3f         0.203
0xf003         0.491
...
# Guest PID    Guest proc runtime (s)
30             1.103
495            0.422
933            0.001
...
```

Figure 1. Sample Identification Data. *The figure shows a notional data set used to identify hidden processes. There is no correlation between VMM and guest process IDs.*

There is no universal, automated technique available for variable selection in multiple regression that is guaranteed to select the best set of variables to include in a model. Stepwise procedures attempt to refine an over-specified or under-specified model iteratively, but often choose bad models. All-possible-subsets regression is guaranteed to choose the best model as long as the number of variables to include is known in advance. As the name implies, all-possible-subsets does this by trying all possible variable combinations of the specified size and maximizing a provided model statistic like the multiple R^2 measure. Unfortunately the cost of all-possible-subsets variable selection grows like $\binom{N}{E}$ where N is the total number of processes and E is our estimate of the number of hidden processes. Since the number of processes to choose from is often large in our environment, this technique is usually far too expensive.

Lycosid uses a simple variable selection heuristic that incorporates what we know about the form of the true model. We know that the coefficients of the variables representing hidden processes should be close to 1.0 and we have an estimate for the total number of hidden processes. Once an initial model incorporating all processes has been fit, those variables corresponding to processes that are obviously not related to the extra observed CPU time are removed from the model. Specifically, variables with negative estimated slopes and variables whose estimated slopes are much greater than 1.0 (e.g., greater than 5 in our implementation) are removed. A new model is then fit using only the remaining variables. Finally, the N variables whose positive relationship to the extra CPU time is strongest are chosen. The strength of a variable's relationship to the extra CPU time is represented by the p-value that results from testing the null hypothesis that the variable's estimated coefficient is zero. Note that we do not attempt to interpret the resulting p-value as a probability related to our identification task. The p-value is simply used to order the variables according to the strength of their relationship to the extra observed CPU time. The top N variables from the ordered list are selected. As in the detection case, we employ a conservative threshold p-value ($\alpha = 1 \times 10^{-5}$) to reduce the chance of false positives, *i.e.*, of incorrectly identifying a process as hidden when it is not. If we do not find N variables with sufficiently small p-values, additional samples are taken and the procedure continues until a configurable upper limit of samples is reached.

4.2 Details

Lycosid obtains CPU time information about processes from both the VMM and from the guest operating system. CPU times for VMM-visible processes are obtained using Antfarm. As in the detection phase, Lycosid invokes documented APIs to obtain and return per-process CPU time information from within the guest.

Samples are obtained from the VMM and from the guest operating system at small random intervals. In our prototype, samples are obtained about once per second on average. A sample consists of a set of process identifiers and the CPU time used by each associated process since the last measurement interval.

Figure 1 shows a notional data set used for identification purposes. Note that Lycosid is unaware of the mapping from guest process IDs to the abstract internal process IDs available within the VMM. No simple method of inferring this mapping currently exists. Otherwise identification would consist of a simple set subtraction operation.

Over time, samples are collected and stored. Once adequate samples have been obtained, a model can be fit and evaluated for hidden process identification. In our current implementation, an initial model is fit once `max(40, number of processes)` samples has been obtained. Up to a maximum of 1000 samples are obtained for use in identification.

4.3 CPU Inflation

The key feature used by our identification algorithm is the CPU time consumed by each process as observed from within the VMM and from within the guest operating system. It is important to note that the identification technique, unlike the detection technique, requires that the hidden process actually runs. *Lycosid* can detect, but not identify a completely idle hidden process.

Lycosid uses a new technique, called *CPU inflation*, that allows it to influence the CPU time used by a process. It is an intrusive technique used only when the passive methods already described fail to reliably identify a hidden process. CPU inflation works by transparently placing patches in guest program code. By forcing processes to run more frequently and more aggressively than they normally would, CPU inflation effectively increases the resolving power of Lycosid's identification techniques.

4.3.1 Details

When control is about to return from the VMM to a guest and CPU inflation is enabled, Lycosid determines the address where execution will resume and places a small patch containing a tight loop at that location. The patch forces the associated process to fully utilize its scheduling quantum until it is removed, effectively maximizing the amount of CPU time used by a process.

Patches are only placed when control returns to user-mode. In our VMM environment, nearly all VMM-to-guest transitions return to kernel-mode. Lycosid must therefore manufacture situations where the VMM returns to user-mode. It accomplishes this by arranging for high-resolution timer interrupts to occur a short time after a return to kernel-mode. The small extra interval allows the operating system to complete its current task (*e.g.*, interrupt processing) and return to user-mode where the guest is ultimately interrupted. An appropriate length for the timer interval can be determined automatically within the VMM by repeatedly increasing the interval until most timer interrupts occur in user-mode. By limiting patches to user-mode code, the normal guest operating system scheduler is free to de-schedule a patched process and the system remains stable.

In our implementation, after a patched process accumulates a certain amount of CPU time, chosen from a configurable, uniformly random interval, the patch is removed and the process is allowed to continue its normal execution. Patches are installed repeatedly according to a configurable patch schedule. Processes that are patched experience reduced performance, but are still allowed to make progress. When CPU inflation is enabled, patching is applied across all running processes. Lycosid enables CPU inflation when the detection module indicates hiding but the identification module is unable to identify the hidden processes.

5. Evasion

We claim that Lycosid is less vulnerable to evasion by guest software than previously presented VMM-based security services. In this section we describe our rationale for the claim and and describe two potential attacks on Lycosid as well as countermeasures.

5.1 Attacking the Trusted View

If a VMM-based security service depends on the correctness of any guest-level component, it is vulnerable to malicious corruption of that component [8]. For example, if a VMM uses the integrity of the guest operating system process list to determine when processes have been hidden, it is subject to evasion when a rootkit based on direct kernel object manipulation corrupts the list. The rootkit leaves the list in a consistent, but incorrect state. A VMM could use additional explicit information about other system components (e.g., thread scheduling queues) to detect inconsistency. The same approach has been taken by guest-level hiding detectors [26], for which there are, unfortunately, malicious work-arounds [1]. In this case, the VMM has no detection advantage over a guest-level tool because the information the VMM uses is fundamentally obtained from the guest.

Unlike previous approaches, the trusted view of operating system processes used by Lycosid is based on implicitly obtained information about observed guest virtual machines. The information is derived from fundamental behaviors of the guest operating system. For example, Lycosid uses process information provided by Antfarm. Antfarm obtains its process information by observing how a guest OS manages its virtual address spaces. To evade Lycosid, an attacker must modify how an infected OS implements a core subsystem (virtual memory) and must do so in a way that remains consistent with its desired user-level view of processes.

5.2 Attacking the Untrusted View

Lycosid depends on an untrusted, user-level process view. One way to attack Lycosid is to manipulate its user-level view. The attack works by *desynchronizing* the untrusted, user-level view used by Lycosid and the user-level view used by a defender to detect unexpected processes (*e.g.*, Windows task manager). In the desynchronization attack, an adversary hides the presence of a malicious process from a defender, but doesn't hide it from Lycosid. In this way Lycosid fails to detect hiding because, from its perspective, no hiding takes place. A defender fails to detect the hidden process because, from their perspective, the malicious process does not exist. Figure 2 shows a conceptual example of the desynchronization attack.

To successfully mount this style of attack, an adversary must be able to reliably identify process enumeration requests made on behalf of Lycosid. In the general case, this task will be difficult because Lycosid uses the same standard APIs to enumerate processes as any other process introspection tool like `ps` or the Windows task manager. Additionally, Lycosid is not limited to using a single tool with a fixed signature to obtain its user-level process view, so an attacker cannot easily rely on a fixed signature database of known Lycosid probe programs. In the same way, there are many different tools that can be used by a defender to enumerate processes (*e.g.*, ps, top, task manager, pslist, tasklist). For the sake of this discussion, however, we will assume an attacker can reliably identify and preferentially handle any Lycosid process enumeration request.

Lycosid is designed to be a part of a larger, comprehensive security monitoring framework. Such a framework would include a process monitoring component that continuously observes the process list and generates an alert when unexpected or suspicious processes are encountered. It is just such a security feature that an attacker hopes to deceive by hiding their malicious processes. The desynchronization attack described above assumes that the process

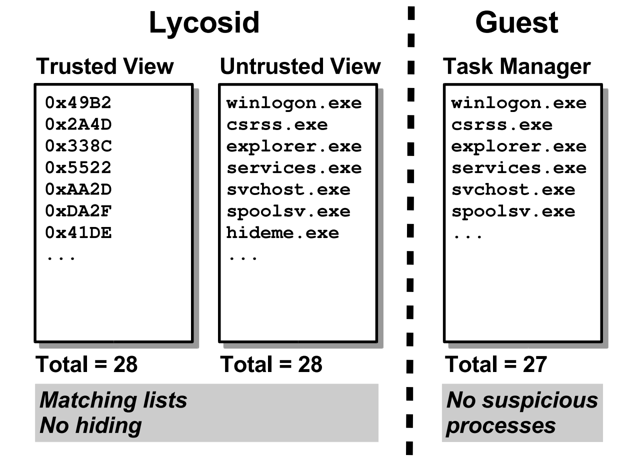

Figure 2. Desynchronization Attack. *The figure demonstrates the desynchronization attack concept against Lycosid hidden process detection.*

view used by the process monitor component is different from the view used by Lycosid. By integrating the process monitor and Lycosid so that they both use the same user-level process view, the opportunity to desynchronize is removed and the attack fails.

In summary, Lycosid is perhaps best described as "differently" subject to gaming and evasion on the part of compromised guests. We believe the effort required to subvert Lycosid while still maintaining a fully consistent outward appearance exceeds that of earlier VMM-based detectors. This is a key feature of VMM-based security services based on implicitly obtained information and raises the bar against malicious process hiding.

6. Implementation

Lycosid is an extension to the Xen [7] VMM. The implementation of Lycosid is split between the Xen hypervisor and user-level programs that run in Xen's privileged control virtual machine.

Antfarm [15] is one hypervisor component. It infers information about guest operating system processes by observing architectural events like page table updates and context switches. Antfarm provides the basis for Lycosid's hidden process detection and identification. CPU inflation is also implemented as a core hypervisor feature. It interposes on Xen's virtual CPU scheduling and shadow page table handling to selectively and safely patch user-level program code. Lycosid adds approximately 850 lines of C code to the hypervisor.

The data collection and analysis components of Lycosid that implement its hidden process detection and identification features are implemented as user-level programs running in a Linux guest virtual machine. They communicate with the hypervisor components of Lycosid via private VMM interfaces that are only available in Xen's privileged control VM. The analysis components are written in python and total approximately 6000 lines of code including statistics libraries and interfaces to libR.so [24], a statistical computing library.

By partitioning Lycosid, only necessary components are added to the hypervisor itself allowing it to remain relatively small, which is a desirable security property. The analysis components are normal user mode programs which can fail and be restarted without compromising the integrity of the whole system. They operate in polled batch mode which removes them from any synchronous critical path and allows them to amortize the cost of their communication with the VMM over many observations.

7. Evaluation

In this section we evaluate the performance of Lycosid's process detection and identification. We want to measure accuracy, timeliness, and runtime overhead. Accuracy is the ability of Lycosid to correctly detect and identify hidden processes measured in terms of false positives and false negatives. Our timeliness experiments measure how long it takes Lycosid to come to its conclusions.

7.1 Experimental Environment

Lycosid is an extension to the Xen [7] VMM version 3.0.3-testing. We use Linux kernel version 2.6.16 in Xen's privileged control virtual machine. We evaluate Lycosid using two guest operating systems. The first is the retail version of Microsoft Windows 2000 Professional. The second is a default installation of Redhat Enterprise Linux 4.3. Both guests run unmodified using Xen's full virtualization support enabled by the Intel virtual machine extensions (VMX) [14]. Our experimental host has a 3.0 GHz Pentium D processor and is configured with 4 GB of system memory. Both privileged and unprivileged virtual machines are allocated 512 MB of memory. The system contains a single Seagate 7200 RPM Barracuda SATA hard disk drive.

7.2 Detection Evaluation

In Section 3 we noted that hidden process detection is complicated by multiple factors. For example, measurements made by the VMM cannot be perfectly synchronized, implicit information can be subtly inaccurate, and unrelated process creation and exit activity make the measurements obtained by Lycosid unstable.

The key variable affecting the ability of Lycosid to detect hidden processes is how much unrelated process creation and exit activity is occurring within the monitored virtual machine. Process creation and exit activity tends to inject variability into the quantities measured by Lycosid and can magnify other, latent sources of variance inherent in the implicit measurement process like lag time [15]. This section evaluates Lycosid's ability to accurately detect a hidden process in spite of these concerns.

7.2.1 Detection with Interference

Our detection experiments evaluate the accuracy and timeliness of Lycosid when detecting a single hidden process. When more than one process has been hidden, the difference between the VMM and user process lists is larger, making detection easier. Hence, detecting a single hidden process is a worst case detection scenario.

To generate process activity we use a synthetic process generator that spawns processes randomly. Harchol-Balter and Downey indicate in their study [11] that process arrivals are burstier than Poisson. We use a pareto distribution with shape parameter $k = 1$ for process inter-arrival times. We control the average rate of process creation by varying the pareto location parameter. This distribution leads to large process creation bursts which stress the detection techniques. The process lifetime distribution described by Harchol-Balter and Downey applies to processes whose lifetime exceeds one second. The arrival rates we use to stress Lycosid, however, are too high to support such long lived processes. As a result, we choose process lifetimes from the uniform distribution on the interval from 0–1 second, which allows our test system to remain stable.

To hide processes under Windows, we use the rootkit tool `fu.exe` and its accompanying device driver `msdirectx.sys` [9]. This tool hides Windows processes by unlinking the target process from the kernel process list. Under Linux we simulate hidden processes by filtering process information in our guest process reporting tool. Unlike `fu.exe`, most recent Linux rootkits hide themselves and manipulate various logging and security features making them inconvenient to use in a research setting.

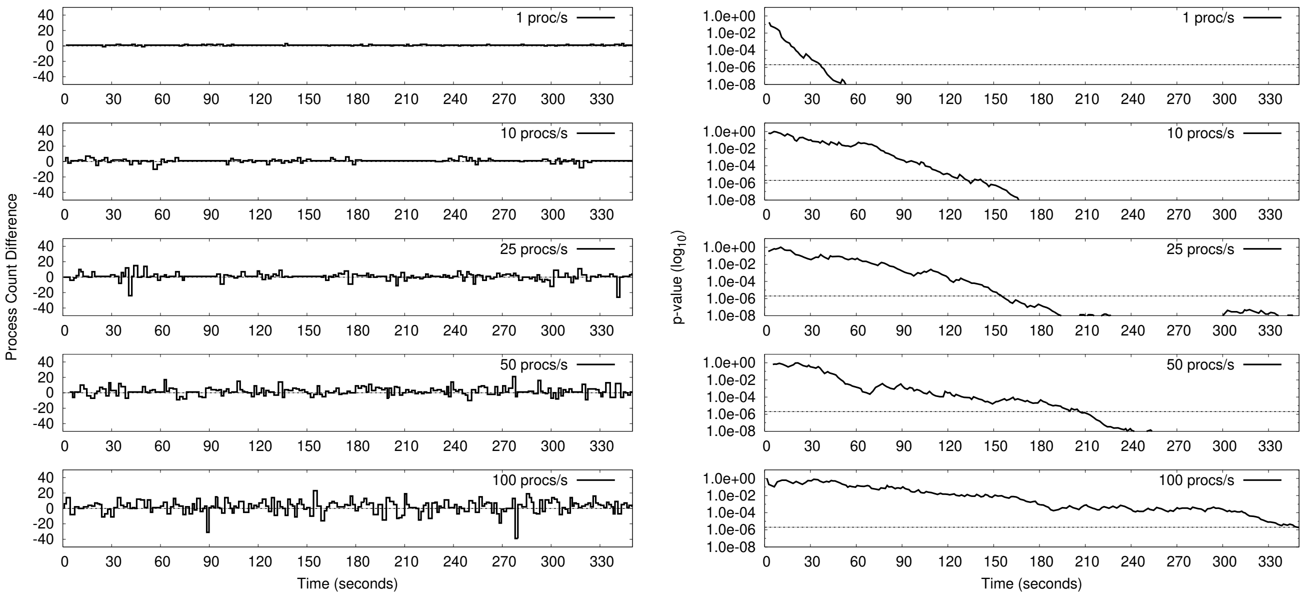

Figure 3. Process Count Difference and Detection Timelines. *The left figure shows a timeline of the difference between the process list length obtained within the VMM and from the guest operating system for various levels of process creation and exit activity. As process activity increases the variability in the measured difference increases. The right figure shows a timeline of the hypothesis test p-values used in the detection process for each of several levels of process creation/exit activity. The p-values approach the detection threshold over time.*

To motivate our use of statistical techniques, the left side of Figure 3 shows how the magnitude of the difference between VMM process count and guest process count used by Lycosid varies over time when the system is subjected to different levels of process creation and exit activity under Windows. As process activity increases from one to an average of 100 processes/second, the variance and magnitude of the difference increase. This characteristic of the detection problem suggests the use of statistical inference techniques to probabilistically determine if hiding is occurring.

The right side of Figure 3 provides intuition about how the p-value resulting from the hypothesis test used by Lycosid incrementally approaches the detection threshold. The test process is hidden immediately when each experiment begins. Detection occurs when the p-value drops below $\alpha = 2 \times 10^{-6}$, which is shown as a dashed horizontal line. In each case an orderly progression toward detection can be seen.

Figure 3 also hints that detection time increases with process activity. To quantify this effect, time to detection was measured for our various process activity levels. The results for Windows and Linux are shown in Figure 4 where the Y-axis reports the time to detection and the X-axis indicates the process activity level. The values shown for each level are the average of 10 trials. The standard deviation of detection time is shown using error bars. Both detection time and its variance increase with process creation and exit activity. In the worst measured cases, under severe process load, Lycosid requires several minutes to detect the hidden process. Since hidden processes are typically long lived (on the order of hours or days) detection times of several minutes are not a real concern. In all of the experiments shown, Lycosid correctly detects the hidden process.

An important output of a positive detection result is an estimate of the number of processes that have been hidden. In the detection experiments described above, a single process was hidden, so, in each case a good estimate will be close to one. Figure 5 shows a summary of the estimated number of hidden processes obtained when a single process has been hidden under both Windows and Linux. When process load is small to moderate, the estimated number of hidden processes is good, leading to a correct identification of one hidden process. Under extreme process creation and exit load, the estimates begin to experience larger error and greater variance. This error may result in falsely identifying a non-hidden process as hidden during the identification phase. However, our conservative p-value identification threshold tends to reduce the chance of false positive identifications. The direction of the error under Windows and Linux is different. Under Windows, Antfarm detects process creation *before* the operating system reports its creation, *i.e.*, process creation lag is negative under Windows. The opposite is true under Linux; Antfarm detects process creation after the OS reports it. High interference and load levels exacerbate the lag under both operating systems leading to larger deviations, but in opposite directions.

7.2.2 False Positives

In addition to reliable detection, it is important that Lycosid not report hidden processes spuriously, *i.e.*, that its false positive rate is small. Our statistical procedure predicts about one false positive result per year. To explore this question empirically, an experiment was performed using a Windows guest in which no process was hidden in our most challenging detection environment (100 process creations and exits/second). An 11 hour timeline from the experiment is shown in Figure 6. As can be seen, no trend toward false detection is apparent and no false detections occur. The experiment does not prove the formal claim of few false positives, but provides empirical support.

7.2.3 Performance Overhead

Lycosid detection is meant to run continuously, so it is important that it impose minimal runtime overhead. To evaluate the overhead of the detection phase of Lycosid we compare the runtimes for three Windows benchmarks when they are run under Lycosid in

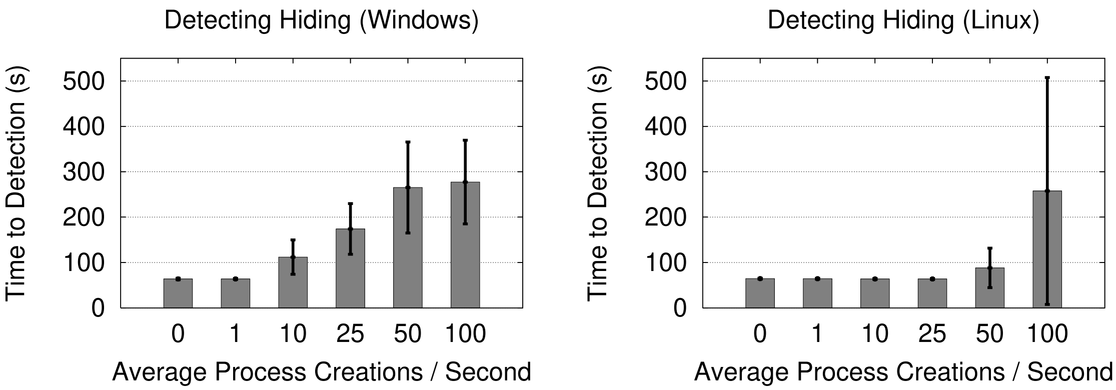

Figure 4. Time to Detection. *The figure shows how the time to detect a hidden process varies for Windows and Linux as process creation and exit activity increases from 0 processes/second to 100 processes/second. The values shown are an average of 10 trials. Error bars show the standard deviation of detection time.*

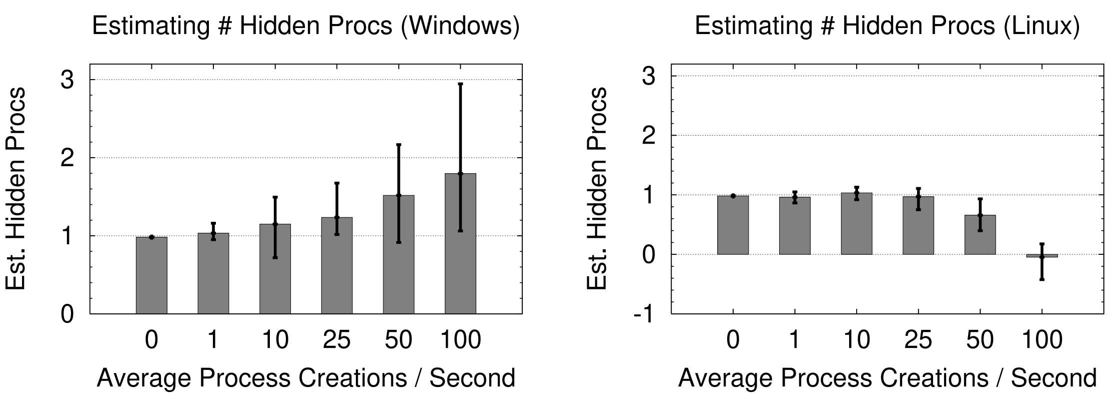

Figure 5. Estimating the Number of Hidden Processes. *The figure shows how the estimate of the number of hidden processes obtained from the detection phase varies for Windows and Linux as process creation and exit activity increases from 0 processes/second to 100 processes/second when a single process has been hidden. The values shown are an average of 10 trials. Error bars show the minimum and maximum hidden process estimate observed.*

Benchmark	Lycosid Runtime	Xen Runtime	% OH
CreateProc	6.551 s	6.222 s	5.3%
MemAlloc	6.803 s	6.565 s	3.6%
Compile	25.386 s	25.210 s	0.7%

Table 1. Detection Runtime Overhead. *The table shows runtimes and overheads for three benchmarks run under Lycosid and under a pristine version of Xen.*

detection mode and when run under an unmodified Xen hypervisor. Table 1 shows the results. Each value is an average of five trials. We observed no significant variance between trials.

Lycosid primarily adds overhead to Xen's shadow page table handling and virtual address space switching. The first two benchmarks spend nearly all of their time performing these two tasks and can be considered worst case scenarios for Lycosid's detection performance. The *CreateProc* benchmark creates and then destroys 1000 processes as quickly as possible. The *MemAlloc* benchmark allocates a 200 MB segment of memory, then touches each page, causing many minor page faults and page table updates. MemAlloc is repeated five times in each trial. Our prototype experiences 5.3% overhead for CreateProc and 3.6% overhead for MemAlloc.

The final benchmark is representative of a more common, but still demanding, workload. It consists of building the bash shell sources using gnu make and gcc. In this case, Lycosid adds a tiny 0.7% overhead.

7.3 Identification Evaluation

In this section we evaluate the ability of the identification algorithm described in Section 4 to identify which processes have been hidden once the detection component provides a positive hiding indicator. As in the evaluation of the detection phase, this evaluation focuses on Lycosid's accuracy and timeliness. In this case, accuracy is Lycosid's ability to correctly identify hidden processes. Our timeliness experiments quantify how long it takes to positively identify the correct hidden processes.

7.3.1 Identification Among Many Running Processes

Our first experiment measures how Lycosid performs when forced to choose among varying numbers of active processes. In the experiments, a number of processes (from 1 to 50) is created. Each of the test processes alternately runs and sleeps. The runtime is chosen randomly from the range 0–500 ms using a uniform distribution. Similarly, a sleep interval is chosen from the interval 0–1000 ms. One of the test processes is hidden using the same techniques described in Section 7.2.1. Experiments were performed with 1, 10,

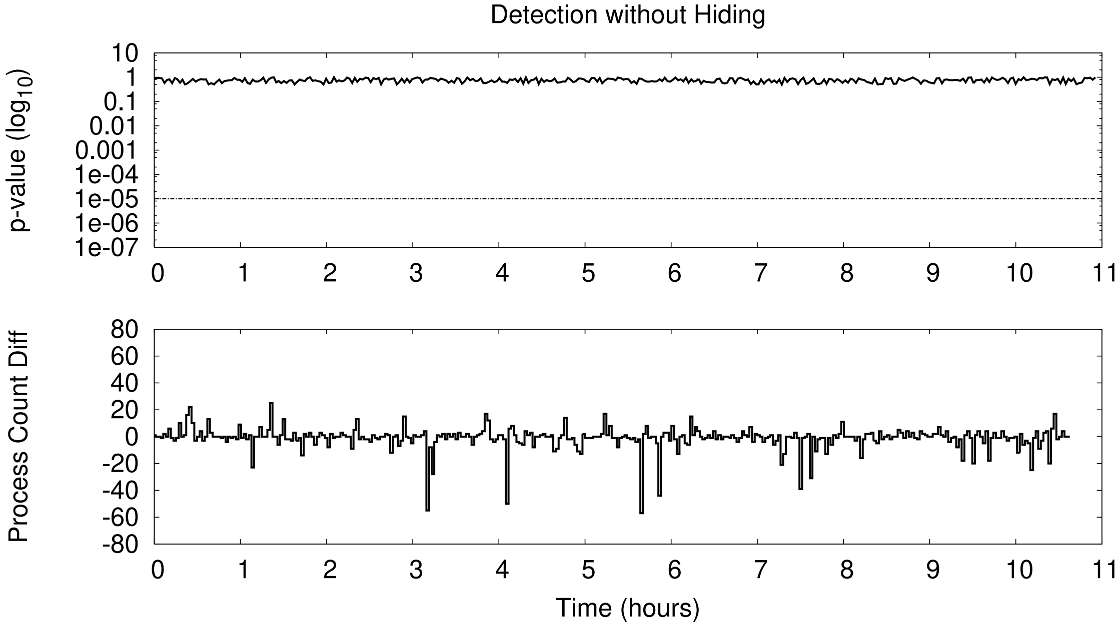

Figure 6. Timeline without Hiding. *The figure shows an approximately 11 hour detection timeline when no processes are hidden and very aggressive process creation/exit activity (100 processes/second) is present. The top graph shows the single-sided hypothesis test p-value. The bottom graph shows the difference between the VMM and guest process counts. No false detections occur.*

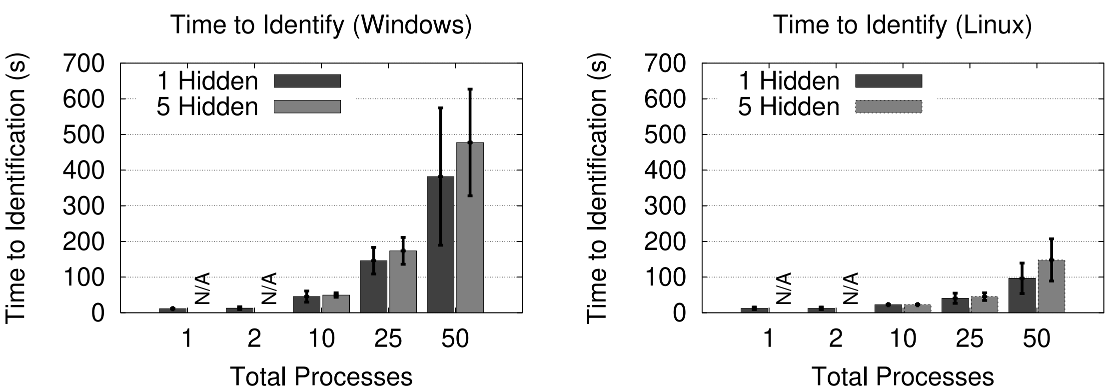

Figure 7. Time to Identification. *The figure shows how the time to identify hidden processes grows as the number of total active processes increases from 1 to 50 processes for both Windows and Linux. The values shown are an average of 10 trials. Lycosid identified the correct hidden processes in all cases on both platforms. Error bars show the standard deviation of identification time. The left bar corresponds to trials in which a single process was hidden. The right bar shows results when 5 processes were hidden.*

25, and 50 total processes. At each level, 10 identification trials were performed. Lycosid correctly identifies the single hidden process in all cases. The time to identify the hidden process for both Windows and Linux guests is shown in Figure 7. The left hand bars show how identification time and standard deviation increase as the number of active processes grows when one process has been hidden. Detection time and variance grow because larger numbers of competing processes decrease the effective runtime of the hidden process. Hence, more samples are required to associate the runtime of the hidden process with the regression response variable in the face of measurement noise.

Hiding multiple processes is a common scenario when an attacker has several distinct tasks to accomplish on a compromised system. Does identification become more difficult when more than one process has been hidden? Our second experiment is similar to the first, but in this case 5 out of the 10, 25, or 50 total processes have been hidden. Again, Lycosid correctly identifies all hidden processes correctly for both platforms. The right hand bars in Figure 7 show that the time to identification grows for the multi-process case, but not significantly. Hence, Lycosid identification is accurate, portable across guest operating systems and applicable in cases where multiple processes have been hidden.

7.3.2 Identifying Mostly Idle Hidden Processes

Our next series of experiments demonstrates that a lower runtime bound exists beneath which Lycosid cannot identify which of several processes is hidden. We then test the ability of CPU inflation to overcome the issue.

Avg. Runtime (s)	Avg Sleep Time (s)	% True ID	% False ID	% No ID
0.25	0.5	**100%**	0%	0%
0.025	0.5	**90%**	0%	10%
0.0025	0.5	**0%**	0%	100%
0.25	**5.0**	**100%**	0%	0%
0.25	**50.0**	**0%**	0%	100%

Table 2. Identification under Reduced Runtime. *The table reports the identification accuracy of Lycosid for a set of experiments in which a single hidden process must be identified among 10 active processes when the hidden process runs exponentially less and less often. As the relative runtime decreases, Lycosid's ability to classify a process as hidden or benign is impaired.*

We first perform two variants of an earlier experiment in which one process is hidden among 10 total active processes under Windows. In each variant we change the runtime of the hidden process along one of two axes. The first axis is busy time, *i.e.*, the time between sleep intervals. The second axis is run frequency, *i.e.*, the length of the sleep intervals. Reducing runtime along either axis decreases the signal-to-noise ratio between hidden process CPU time and the measurement error experienced by Lycosid. The effect is to make identification more challenging.

In the first set of experiments we reduce hidden process busy time by factors of 10 and measure the ability of Lycosid to identify the hidden process. In the second round of experiments we increase the sleep interval by factors of 10 and again evaluate if Lycosid can identify the hidden process. Table 2 lists the runtime parameters for the hidden process in each experiment and the percentage of 10 trials in which Lycosid successfully identifies the single hidden process.

When the busy time is reduced from earlier experiments by a factor of 10 Lycosid correctly identifies the hidden processes in only 9 of 10 trials. After reducing the runtime by a factor of 100, no process exceeds the identification threshold p-value before the implementation sample limit of 1000 is reached; hence, no process is identified as hidden. When the sleep time increases by a factor of 10 or 100, none of 10 trials produces a positive hidden process identification. Note that in no case do false positives occur, *i.e.*, no innocent processes are accused of being hidden. We see, however, that if a hidden process runs for limited periods, even if it runs regularly, or if a hidden process runs infrequently, Lycosid cannot identify it properly. Even in these cases, however, Lycosid correctly detects that process hiding is taking place.

Table 3 shows the results of applying CPU inflation to identification tasks in which the hidden process runs for short periods of time or rarely runs. Our evaluation shows that CPU inflation enables Lycosid to identify processes whose average busy time is as low as 250 μs. The table also shows that even when a hidden process runs relatively rarely (*e.g.*, once every 500 seconds on average) CPU inflation makes the hidden process identifiable by Lycosid. Finally, when the hidden process's average sleep time exceeds the amount of time over which Lycosid makes observations (once every 5000 seconds vs. approximately 1000 seconds of observation time in this experiment) Lycosid is naturally unable to reliably identify the hidden process. Our evaluation shows that CPU inflation is a powerful tool that significantly extends the set of hidden processes that Lycosid can reliably identify.

8. Related Work

Cross-view validation for hiding detection has been studied and variously implemented in user applications [5], within the operating system kernel [30], inside a virtual machine monitor [10], and using dedicated coprocessor hardware [23]. The key aspect of cross-view validation that differentiates these efforts is the mechanism used to obtain the low-level, trusted view of the resource of interest.

Garfinkel *et al.*, have shown the value of VMM-level cross-view validation for detecting hidden processes with VMI [10]. VMI uses explicit operating system debugging information to locate and interpret private kernel data types at runtime. This insight into operating system data structures is used to obtain a trusted view of the guest operating system process list. Lycosid extends the VMI concept by using only implicitly obtained guest information within a VMM. No implementation details are required. This allows Lycosid to be deployed in situations where version and patch-level-specific debugging information is unavailable or inconvenient to maintain as a system is patched and upgraded.

Instead of a guest kernel-level view of the process list as used by VMI, Lycosid uses a true VMM-level process view. The VMM view, which is based on observations of guest virtual address spaces, should be more challenging for malicious software to manipulate. More fundamental aspects of the execution of a hidden process would need to be altered to enable evasion, *e.g.*, how a process uses virtual memory and how the operating system accounts runtime to processes.

Many systems employ statistical techniques to infer behavior, to provide input to control algorithms, and to implement security classifiers. For example, MS Manners [6] uses hypothesis testing to regulate the scheduling of low-priority background processes and reduce their performance impact on high priority foreground jobs. Jung *et al.* [18] probabilistically determine whether remote hosts are conducting port scanning using sequential hypothesis testing techniques [29]. One of Lycosid's key features is its use of statistical inference techniques to overcome the noise fundamental to the implicit information it uses.

9. Conclusion

Lycosid is a novel VMM-based hidden process detection and identification service. The key difference between Lycosid and previous VMM-based hidden process detectors is Lycosid's use of implicitly obtained information about the guest operating systems it monitors. Implicit information decouples Lycosid from the guest OS and allows it to take better advantage of its placement within a VMM. For example, Lycosid does not depend on the consistency of private guest OS data structures, so it is less vulnerable to guest-initiated evasion attacks. Similarly, Lycosid does not depend on guest OS implementation details, so it can be portable across very different operating systems.

Using implicitly obtained information within a VMM can be challenging because it is often noisy or wrong. Lycosid provides an accurate and reliable service in spite of its noisy inputs by using statistical inference techniques like hypothesis testing and regression to trade detection and identification time for accuracy.

In our evaluation, Lycosid correctly detected process hiding in each of hundreds of trials. Identification is similarly robust except in cases where a hidden process does not run long enough or fre-

Avg. Runtime (s)	Avg Sleep Time (s)	% True ID	% False ID	% No ID
0.025	0.5	**100%**	0%	0%
0.0025	0.5	**100%**	0%	0%
0.00025	0.5	**100%**	0%	0%
0.025	**5.0**	**100%**	0%	0%
0.025	**50.0**	**100%**	0%	0%
0.025	**500.0**	**100%**	0%	0%
0.025	**5000.0**	**20%**	0%	80%

Table 3. Effect of CPU Inflation. *The table shows how CPU inflation can help make hidden processes that run relatively little identifiable by Lycosid. In the experiments, a single hidden process must be identified among 10 active processes when the hidden process runs very little or infrequently. CPU inflation forces the hidden process to run more, providing Lycosid with the information it needs to make a positive identification. When average sleep time exceeds the maximum sample period, Lycosid naturally fails to reliably identify all hidden processes.*

quently enough. Lycosid uses a new technique, called CPU inflation, that can force some difficult to identify processes into an execution regime in which a hidden process can be positively identified.

The detection components of Lycosid, which are designed to run continuously, impose a very small runtime overhead. In a worst-case performance scenario, we measured less than 6% overhead. For a more typical, process-intensive workload, Lycosid imposes a mere 0.7% penalty.

Research interest in VMM-based security services is already strong [8, 10, 17] and commercial VMM-based security products from industry leaders are coming soon [13]. This paper describes real experiences using implicitly obtained information effectively within VMM-based services. Our results show that, while challenging, using implicitly obtained guest information can improve the capabilities and safety properties of such services.

Acknowledgments

We would like to thank the anonymous reviewers for their thoughtful comments and suggestions. This research is sponsored by the Sandia National Laboratories Doctoral Studies Program, by NSF CCR-0133456, ITR-0325267, CNS-0509474, and by generous donations from Network Appliance and EMC.

References

[1] 90210. Bypassing klister 0.4 with no hooks or running a controlled thread scheduler. hi-tech.nsys.by/33/.

[2] J. Butler, J. L. Undercoffer, and J. Pinkston. Hidden processes: The implication for intrusion detection. In *Proceedings of the 2003 IEEE Workshop on Information Assurance*, pages 116 – 121, June 2003.

[3] J. Clemens. Knark: Linux kernel subversion. www.sans.org/ resources/ idfaq/ knark.php.

[4] B. Cogswell and M. Russinovich. Pslist. www.sysinternals.com.

[5] B. Cogswell and M. Russinovich. Rootkit revealer. www.sysinternals.com.

[6] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 247–260, Kiawah Island Resort, South Carolina, December 1999.

[7] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.

[8] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[9] fuzen_op. fu.exe and msdirectx.sys. www.rootkit.com/.

[10] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[11] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Comput. Syst.*, 15(3):253–285, 1997.

[12] Holy Father. HackerDefender. hxdef.org.

[13] Intel and Symantec Corp. Symantec virtual security solution and pcs with intel vpro technology. http://www.intel.com/ business/ casestudies/ symantec_solutions_brief.pdf.

[14] Intel Corporation. Vtx specification. developer.intel.com, 2005.

[15] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In *Proceedings of the USENIX Annual Technical Conference (USENIX '06)*, Boston, Massachusetts, June 2006.

[16] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, San Jose, California, October 2006.

[17] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 91–104. ACM Press, 2005.

[18] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In *IEEE Symposium on Security and Privacy 2004*, Oakland, CA, May 2004.

[19] P. Karger, M. Zurko, D. Bonin, A. Mason, and C. Kahn. A retrospective on the VAX VMM security kernel. In *IEEE Transactions on Software Engineering*, volume 17, pages 1147–1165, November 1991.

[20] Microsoft. Windows malicious software removal tool. www.microsoft.com.

[21] T. Miller. t0rn rootkit. www.ossec.net/ rootkits/ studies/ t0rn.txt.

[22] R. Naraine. Microsoft: Stealth Rootkits Are Bombarding XP SP2 Boxes. www.eweek.com/article2/0,1895,1896605,00.asp.

[23] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, August 2004.

[24] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. ISBN 3-900051-07-0.

[25] F. L. Ramsey and D. W. Schafer. *The Statistical Sleuth: A Course in Methods of Data Analysis*. Duxbury Press, Boston, MA, 2nd edition, 2002.

[26] J. Rutkowska. klister. www.invisiblethings.org/ tools/ klister-0.4.zip.

[27] SANS Institute. Subseven trojan v 1.1. www.sans.org/ resources/ idfaq/ subseven.php.

[28] sd and devik. Suckit. Phrack #58, article 0x07.

[29] A. Wald. *Sequential Analysis*. John Wiley & Sons, Inc., New York, NY, 3rd edition, September 1952.

[30] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting stealth software with strider ghostbuster. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2005)*, pages 368–377, June 2005.